ARCADE SERIES

THE BOOK OF CP-SYSTEM

FABIEN SANGLARD

MEMSET PUBLICATIONS

Foreword

There used to be a time when video game enthusiasts could only experience the very best in places called ”arcades”.

In the early ’90s, 16-bit home consoles such as the Super Nintendo, the Sega Genesis, or the NEC PC Engine were ramping up in terms of horsepower. However, they were a far cry from the hardware found in coin-operated ”Amusement Machines”.

Nicknamed ”coin-ops”, these cabinets ran video games featuring multitudes of huge sprites covering the whole screen, beautiful colors, digitized sounds, and engaging high quality music. These machines were in a league of their own.

Accessing arcades was an adventure in itself. Quarters had to be gathered, means of transportation acquired, and paper maps studied. Some carpooled while others used their bikes. Lucky ones had ”amusement venues” dedicated to video games in their hometown while others found themselves in a dirty pub surrounded by adults who did not seem to have much magic happening in their lives.

Amount of play time was directly correlated with skill level. Coins were spent carefully, after having studied other people’s techniques. The only certainty resulting from the expedition was a day ending with empty pockets.

Despite all these obstacles, video game connoisseurs found the attraction irresistible. Players of all ages and origins gravitated to the same places in order to follow their passion.

Rows of lined up cabinets created a highly competitive environment where publishers only had a few seconds to catch a player’s attention and, most importantly, their quarters. It was during this time that a young company named Capcom managed to rise above the competition, seemingly producing one masterpiece after another, and turn itself into an icon.

The history of Capcom and the genesis of Street Fighter II, Ghouls 'n Ghosts, and Final Fight belongs in history books. Unfortunately when I started researching the topic, I found little to satisfy my curiosity and next to nothing about the engineering side of things.

The fierce rivalry between publishers warranted extreme secrecy. Artists, programmers, and designers were only credited with their nicknames in order to avoid poaching. As for the hardware powering Capcom's titles, nothing ever officially transpired except for a code name, **CP-System**.

This book attempts to shed some light on the mystery platform. It is an engineering love letter to the machine that enabled Capcom's tremendous success.

– Fabien Sanglard
Occasional Link to the Past

Sunnyvale, CA
December, 2022

Contents

Enemies and their Discontents

ACONT

This is the bit that I knew would take me ages to write and get glitch free, and the bit that is absolutely necessary to the functioning of the game. The module ACONT is essentially an interpreter for my own 'wave language', allowing me to describe, exactly, an attack wave in about 50 bytes of data. The waves for the first part of IRIDIS are in good rollicking shoot-'em-up style, and there have to be plenty of them. There are five planets and each planet is to have twenty levels associated with it. It's impractical to write separate bits of code for each wave; even with 64K you can run outta memory pretty fast that way, and it's not really necessary coz a lot of stuff would be duplicated. Hence ACONT.

— Jeff Minter's Development Diary in Zzap Magazine[2]

The bits and bytes that define the behaviour and appearance of wave after wave of Iridis Alpha's enemy formations - twenty across each of the five planets giving on hundred in all - takes up relatively little space given the sheer variety of adversaries the player faces.

1.0.1 You're a Waste of Space

Each 'wave' of enemies is defined by a 40 byte data structure, not 50 bytes as Minter initially suggested in his development diary. There's a little bit of waste going on in here too, bytes 10 to 14 are unused, while `Byte 15` is only ever set (to `$01`) by the wave data structure that describes the default explosion behaviour for enemy ships.

As we can see here, the sole purpose of `Byte 15` is to determine whether a new set of wave data needs to be loaded. This makes sense, once the animation is finished we'll need to load a new enemy ship. Still, you can't help thinking there might have been a way that didn't waste 99 bytes!

```
        LDA upperPlanetAttackShipYPosUpdated,X
        BEQ b4D98
        LDA #$00
        STA upperPlanetAttackShipYPosUpdated,X
        ; The 2nd stage of wave data for this enemy.
        LDY #$19
        LDA (currentShipWaveDataLoPtr),Y
        BEQ b4D98
        DEY
        JMP UpdateWaveDataPointersForCurrentEnemy

b4D98   LDA upperPlanetAttackShipYPosUpdated2,X
        BEQ No3rdWaveData
        LDA #$00
        STA upperPlanetAttackShipYPosUpdated2,X
        ; The 3rd stage of wave data for this enemy.
        LDY #$1B
        LDA (currentShipWaveDataLoPtr),Y
        BEQ No3rdWaveData
        DEY
```

Listing 1.1: "Routine for Animating Enemy Sprites"

And actually, it's more than that because as we shall see the `ACONT` 40-byte data structure is defined more than once per wave. Separate instances are defined for later phases of the enemy ship, such as when it is first hit. Early examples of this in the game are the 'spinning rings' you get when you hit an enemy in the first level.

In all there are 200 instances of the `ACONT` data structure: 100 defining each of the enemy waves and another 100 defining the subsequent behaviour of the ships when hit. There isn't a one-to-one mapping here either - many of the effects are reused across levels and as we shall see there can be multiple stages in an enemy's lifecycle.

So there's already 1000 bytes or 1 kilobyte of wasted space in the level data due to bytes that are never or rarely used. That's out of a total of 8 kilobytes actually used.

Shocking stuff. Awful.

1.0.2 And You're a Waste of Space

`Bytes 33 - 34` seem to be left in an unfinished state. Wave 12 on Planet 5 has both populated, `flowchartArrowAsExplosion` is the only other wave that has anything in either byte, in this case `$60` in `Byte 33`.

This another 200 wasted bytes it seems but it seems these bytes have some game logic attached and when you look at what that logic is doing it seems broken.

```
        BNE b4D7F
```

```
EnergyUpdateTopPlanet
        LDA extraAmountToDecreaseEnergyByTopPlanet
        BNE b4D7F
        ; Y is still $23.
        LDA (currentShipWaveDataLoPtr),Y
        JSR AugmentAmountToDecreaseEnergyByBountiesEarned
        STA extraAmountToDecreaseEnergyByTopPlanet
b4D7F   LDY #$1E
        JMP UpdateWaveDataPointersForCurrentEnemy
        ; Returns

;------------------------------------------------------------------
```

Listing 1.2: ”Routine for Animating Enemy Sprites”

When Byte 34 (`$21`) is populated (and fire has not been pressed) the game will attempt to load a set of wave date from Bytes 33 and 34:

```
        BEQ No3rdWaveData
        LDA #$00
        STA upperPlanetAttackShipYPosUpdated2,X
        ; The 3rd stage of wave data for this enemy.
        LDY #$1B
        LDA (currentShipWaveDataLoPtr),Y
        BEQ No3rdWaveData
        DEY
        JMP UpdateWaveDataPointersForCurrentEnemy

No3rdWaveData
        LDA joystickInput
        AND #$10
        BNE Byte34IsZero
        ; Check if we should load extra stage data for this enemy.
        ; FIXME: When this is set it would incorrectly expect there
        ; to be a hi/lo ptr in $20 and $21, when there isn't.
        LDY #$21
        LDA (currentShipWaveDataLoPtr),Y
```

Listing 1.3: ”Routine for Animating Enemy Sprites”

In the case of the data for Planet 5 Level 12 this translates to whatever is at `$1488`. As it happens this is the address of the frequency data used in the title screen’s music. So effectively pretty random data:

```
.include "dna.asm"


; This is the frequency table containing all the 'notes' from
; octaves 4 to 8. It's very similar to:
;  http://codebase.c64.org/doku.php?id=base:ntsc_frequency_table
; The 16 bit value you get from feeding the lo and hi bytes into
; the SID registers (see PlayNoteVoice1 and PlayNoteVoice2) plays
; the appropriate note. Each 16 bit value is based off a choice of
; based frequency. This is usually 440hz, but not here.

                  ;     C   C#  D   D#  E   F   F#  G   G#  A   A#  B
titleMusicHiBytes .BYTE $08,$08,$09,$09,$0A,$0B,$0B,$0C,$0D,$0E,$0E,$0F  ; 4
                  .BYTE $10,$11,$12,$13,$15,$16,$17,$19,$1A,$1C,$1D,$1F  ; 5
                  .BYTE $21,$23,$25,$27,$2A,$2C,$2F,$32,$35,$38,$3B,$3F  ; 6
                  .BYTE $43,$47,$4B,$4F,$54,$59,$5E,$64,$6A,$70,$77,$7E  ; 7
```

Listing 1.4: ”Routine for Animating Enemy Sprites”

Clearly, no one has ever reached level 12 in planet 5!

1.0.3 Clever Business

> You pass the interpreter data, that describes exactly stuff like: what each alien looks like, how many frames of animation it uses, speed of that animation, colour, velocities in X— and Y— directions, accelerations in X and Y, whether the alien should 'home in' on a target, and if so, what to home in on; whether an alien is subject to gravity, and if so, how strong is the gravity; what the alien should do if it hits top of screen, the ground, one of your bullets, or you; whether the alien can fire bullets, and if so, how frequently, and what types; how many points you get if you shoot it, and how much damage it does if it hits you; and a whole bunch more stuff like that. As you can imagine it was a fairly heavy routine to write and get debugged, but that's done now; took me about three weeks in all I'd say.
>
> — Jeff Minter's Development Diary in Zzap Magazine[2]

The level data does actually define some of this stuff. It does so by making heavy use of a simple but clever trick that in its way is very specific to 8-bit assembly programming: storing references to other data structures as a pair of bytes. We've discussed the way this works previously but we'll try again briefly here as it won't do any harm.

The 40-byte data structure that defines the default explosion animation (and behaviour, so far as it goes) is stored at position `$18C8` while the game is running. To use this explosion data when an enemy is killed, bytes 31 and 32 of the data structure contain the values `$C8,$18`.

When an enemy is hit, the game routine responsible for figuring out what to do next with it looks at bytes 31 and 32 and loads in the data structure at the address given by combining `$18` and `$C8` as the 'new' wave data that defines how that enemy ship will now behave. Since the data structure at `$18C8` basically says: animate an explosion sprite and don't move anywhere that is exactly what the enemy ship now does.

Here's the explosion data structure, which we've labelled `defaultExplosion` in our disassembly, in the first twenty bytes or so of it's gory detail:

```
defaultExplosion = $18C8
        ; Byte 1 (Index $00): An index into colorsForAttackShips that applies a
        ; color value for the ship sprite.
        .BYTE $07
        ; Byte 2 (Index $01): Sprite value for the attack ship for the upper planet.
        ; Byte 3 (Index $02): The 'end' sprite value for the attack ship's animation
        ; for the upper planet.
        .BYTE EXPLOSION_START,EXPLOSION_START + $03
        ; Byte 4 (Index $03): The animation frame rate for the attack ship.
        .BYTE $03
        ; Byte 5 (Index $04): Sprite value for the attack ship for the lower planet.
        ; Byte 6 (Index $05): The 'end' sprite value for the ship's lower planet animation.
        .BYTE EXPLOSION_START,EXPLOSION_START + $03
        ; Byte 7 (Index $06): Whether a specific attack behaviour is used.
        .BYTE $00
        ; Byte 8 (Index $07): Lo Ptr for an unused attack behaviour
        ; Byte 9 (Index $08): Hi Ptr for an unused attack behaviour
        .BYTE <nullPtr,>nullPtr
        ; Byte 10 (Index $09): Lo Ptr for an animation effect? (Doesn't seem to be used?)
        ; Byte 11 (Index $0A): Hi Ptr for an animation effect (Doesn't seem to be used?)?
        .BYTE <nullPtr,>nullPtr
        ; Byte 12 (Index $0B): some kind of rate limiting for attack wave
        .BYTE $00
        ; Byte 13 (Index $0C): Lo Ptr for a stage in wave data (never used).
        ; Byte 14 (Index $0D): Hi Ptr for a stage in wave data (never used).
        .BYTE <nullPtr,>nullPtr
        ; Byte 15 (Index $0E): Controls the rate at which new enemies are added?
        .BYTE $01
        ; Byte 16 (Index $0F): Update rate for attack wave
        .BYTE $0D
        ; Byte 17 (Index $10): Lo Ptr to the wave data we switch to when first hit.
        ; Byte 18 (Index $11): Hi Ptr to the wave data we switch to when first hit.
        .BYTE <nullPtr,>nullPtr
        ; Byte 19 (Index $12): X Pos movement for attack ship.
        .BYTE $80
        ; Byte 20 (Index $13): Y Pos movement pattern for attack ship.
```

Listing 1.5: ”Routine for Animating Enemy Sprites”

We can see the first 7 bytes are concerned with the appearance and basic behaviour of the enemy. Bytes 2 and 3 define the sprite used for display on the upper planet, Bytes 5 and 6 for the lower planet. The reason there's two in each case is because they are describing the start and end point of the sprite's animation. The game will display `EXPLOSION_START $ED`) first, then cycle through the next two sprites until it reaches `EXPLOSION_START` + 3 (`$F0`).

1.0.4 Sprites Behaving Badly

We can see this in action in `AnimateAttackShipSprites`. When this routine runs Byte 4 has been loaded to `upperPlanetAttackShipInitialFrameRate` for the upper planet and `lowerPlanetAttackShipInitialFrameRate` for the lower planet. This routine is cycling through the sprites given by Byte 2 as the lower limit and Byte 3 as the upper limit. This is what the animation consists of: an animation effect achieved by changing the sprite from one to another to create a classic animation effect.

```
UpperBitSetOnXPosMovement
        ; This creates a decelerating effect on the attack ship's movement.
        ; Used by the licker ship wave in planet 1 for example.
        EOR #$FF
        STA attackShipOffsetRate
        INC attackShipOffsetRate
        LDA upperPlanetAttackShip2XPos,X
        SEC
        SBC attackShipOffsetRate
        STA upperPlanetAttackShip2XPos,X
        BCS DecrementXPosFrameRateLowerPlanet

        LDA upperPlanetAttackShip2MSBXPosValue,X
        EOR attackShip2MSBXPosOffsetArray,X
        STA upperPlanetAttackShip2MSBXPosValue,X

DecrementXPosFrameRateLowerPlanet
        DEC lowerPlanetXPosFrameRateForAttackShip - $01,X
        BNE ReturnFromUpdateXYPositions
        LDA lowerPlanetInitialXPosFrameRateForAttackShip - $01,X
        STA lowerPlanetXPosFrameRateForAttackShip - $01,X
        LDA xPosMovementForLowerPlanetAttackShip - $01,X
        BMI UpperBitSetOnXPosMovementLowerPlanet

        CLC
        ADC lowerPlanetAttackShip3XPos ,X
        STA lowerPlanetAttackShip3XPos ,X
        BCC ReturnFromUpdateXYPositions
        LDA lowerPlanetAttackSHip3MSBXPosValue ,X
        EOR attackShip2MSBXPosOffsetArray ,X
        STA lowerPlanetAttackSHip3MSBXPosValue ,X
        JMP ReturnFromUpdateXYPositions

UpperBitSetOnXPosMovementLowerPlanet
```

Listing 1.6: ”Routine for Animating Enemy Sprites”

Byte 2 (loaded to `upperPlanetAttackShipAnimationFrameRate` comes into play here. It’s decremented and as long as it’s not zero yet the animation is skipped, execution jumps forward to `AnimateLowerPlanetAttackShips`:

```
        LDA upperPlanetAttackShip2MSBXPosValue ,X
        EOR attackShip2MSBXPosOffsetArray ,X
```

Listing 1.7: ”Routine for Animating Enemy Sprites”

If it is zero, it instead gets reset to the initial value from Byte 2 (stored in `upperPlanetAttackShipInitialFra`
and the current sprite for the enemy ship is incremented to point to the next ’frame’ of the ship’s animation:

```
DecrementXPosFrameRateLowerPlanet
        DEC lowerPlanetXPosFrameRateForAttackShip - $01,X
        BNE ReturnFromUpdateXYPositions
```

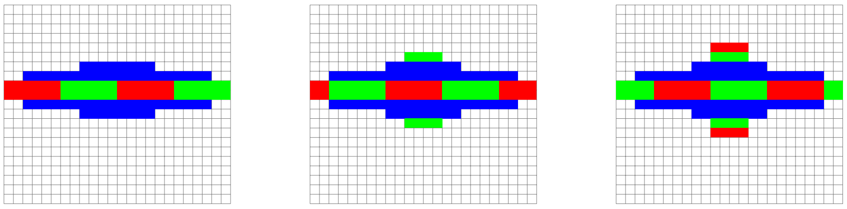
Listing 1.8: ”Routine for Animating Enemy Sprites”

Next we check if we’ve reached the end of the animation by checking the value of Byte 3 (loaded to `upperPlanetAttackShipSpriteAnimationEnd`). If so, we reset `upperPlanetAttackShip2SpriteValue` to the value initially loaded from Byte 2 - and that is what will be used to display the ship the next time we pass through to animate the ship:

```
        LDA lowerPlanetInitialXPosFrameRateForAttackShip - $01,X
        STA lowerPlanetXPosFrameRateForAttackShip - $01,X
        LDA xPosMovementForLowerPlanetAttackShip - $01,X
        BMI UpperBitSetOnXPosMovementLowerPlanet

        CLC
        ADC lowerPlanetAttackShip3XPos ,X
        STA lowerPlanetAttackShip3XPos ,X
```

Listing 1.9: ”Routine for Animating Enemy Sprites”



(a) FLYING_SAUCER1 **(b)** FLYING_SAUCER2 **(c)** FLYING_SAUCER3

Figure 1.1: The sprites used to animate the 'UFO' in the first level.

1.0.5 Enemy Movement

Enemy movement is controlled by two parameters in each direction: the number of pixels to move in one go and the number of cycles to wait between each movement. So for movement in the horizontal (or X direction) `Byte 19` controls the number of pixels to move at once, while `Byte 21` controls the number of cycles to wait between each movement. The same applies to `Byte 20` and `Byte 22` for the vertical (or Y direction).

If we look at Byte 19 and Byte 21 for Level 1 we can see that the the fast lateral movement of the 'UFO's is implemented by a relatively high value of $06 for the number of pixels it moves at each step while the interval between steps is relatively low ($01). Meanwhile the more gradual up and down movement is implemented by a value of $01 in Byte 20 and Byte 22.

For the second level ('bouncing rings') the horizontal movement is more constrained (Byte 19 is $00) while the vertical movement is more extreme (Byte 20 is $24) - achieving the bouncing effect.

Level	Byte 7	Byte 19	Byte 20	Byte 21	Byte 22
1	$00	$06	$01	$01	$01
2	$00	$00	$24	$02	$01
3	$00	$FA	$01	$01	$02

Byte 7 : Whether a specific attack behaviour is used.
Byte 19: X Pos movement for attack ship.
Byte 20: Y Pos movement pattern for attack ship.
Byte 21: X Pos Frame Rate for Attack ship.
Byte 22: Y Pos Frame Rate for Attack ship.

Movement data for the first three levels.

The horizontal movement for level three is $FA, which would make you think the enemies must be moving horizontally extremely quickly. In fact, when the high bit is set a special behaviour is invoked:

```
        LDA lowerPlanetAttackShip2YPos,X
        CMP #$99
```

Listing 1.10: "From `UpdateAttackShipsXAndYPositions.` "

When the upper bit is set (e.g. $FC,$80) on the value loaded to the accumulator by `LDA` then `BMI` will return true and jump to `UpperBitSetOnXPosMovement`.

```
        STA upperPlanetAttackShipYPosUpdated + $07,X
b7DAF   LDA #$00
        STA yPosMovementForLowerPlanetAttackShips - $01,X
b7DB4   DEC upperPlanetXPosFrameRateForAttackShip - $01,X
        BNE DecrementXPosFrameRateLowerPlanet

        LDA upperPlanetInitialXPosFrameRateForAttackShip - $01,X
        STA upperPlanetXPosFrameRateForAttackShip - $01,X

        LDA xPosMovementForUpperPlanetAttackShip - $01,X
        BMI UpperBitSetOnXPosMovement

        CLC
        ADC upperPlanetAttackShip2XPos,X
        STA upperPlanetAttackShip2XPos,X
```

Listing 1.11: "From `UpdateAttackShipsXAndYPositions.` "

This first line `EOR #$FF` performs an exclusive-or between Byte 19 in the `Accumulator` ($FA) and the value $FF. An exclusive-or, remember, is a bit by bit comparison of two bytes which will set a bit in the result if an only if the bit in one of the values is set but the other is not:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$FF	1	1	1	1	1	1	1	1
$FA	1	1	1	1	1	0	1	0
Result	0	0	0	0	0	1	0	1

X-OR'ing $FF and $FA gives $05.

This result is stored in `attackShipOffsetRate`:

```
```

Listing 1.12: "From `UpdateAttackShipsXAndYPositions.` "

Incremented:

```
        LDA upperPlanetInitialXPosFrameRateForAttackShip - $01,X
```

Listing 1.13: "From `UpdateAttackShipsXAndYPositions.` "

And then subtracted from the enemy's X position:

```
        STA upperPlanetXPosFrameRateForAttackShip - $01,X

        LDA xPosMovementForUpperPlanetAttackShip - $01,X
        BMI UpperBitSetOnXPosMovement
```

Listing 1.14: "From `UpdateAttackShipsXAndYPositions.` "

The net result is a deceleration effect. This is observed in the way the licker ship wave will accelerate out to the center before dialling back again.

What is going on with Byte 7?

Byte 7 comes into play when setting the initial Y position of a new enemy. This initial vertical position is random, but subject to some adjustment:

```
previousAttackShipIndexTmp = tmpPtrLo
SetInitialRandomPositionsOfEnemy
        LDY previousAttackShipIndex
        LDA indexForActiveShipsWaveData,X
        TAX
        LDA attackShipsMSBXPosOffsetArray + $01,X
        STA upperPlanetAttackShipsMSBXPosArray + $01,Y
        JSR PutRandomByteInAccumulatorRegister
        AND #$7F
        CLC
        ADC #$20
        STA upperPlanetAttackShipsXPosArray + $01,Y

        ; Are we on the upper or lower planet?
        TYA
        AND #$08
        BNE SetInitialRandomPositionLowerPlanet

SetInitialRandomPositionUpperPlanet
        JSR PutRandomByteInAccumulatorRegister
        AND #$3F
        CLC
        ADC #$40
        STA upperPlanetAttackShipsYPosArray + $01,Y

        STY previousAttackShipIndexTmp
        ; Byte 7 ($06): Usually an update rate for the attack
    ships.
        LDY #$06
        LDA (currentShipWaveDataLoPtr),Y
        BNE ReturnFromLoadingWaveDataEarly

        ; Byte 9 ($08): Default initiation Y position for the
    enemy.
        LDY #$08
        LDA (currentShipWaveDataLoPtr),Y
        BEQ ReturnFromLoadingWaveDataEarly

        LDA #$6C
        LDY previousAttackShipIndexTmp
        STA upperPlanetAttackShipsYPosArray + $01,Y

ReturnFromLoadingWaveDataEarly
        RTS

SetInitialRandomPositionLowerPlanet
```

```
        JSR PutRandomByteInAccumulatorRegister
        AND #$3F
        CLC
        ADC #$98
        STA upperPlanetAttackShipsYPosArray + $01,Y

        STY previousAttackShipIndexTmp
        ; Byte 7 ($06): Determines if the inital Y Position of
    the ship is random or uses a default.
        LDY #$06
        LDA (currentShipWaveDataLoPtr),Y
        BNE ReturnFromLoadingWaveData

        ; Byte 9 ($08): A Hi-Ptr to wave data normally but
    treated here .
        LDY #$08
        LDA (currentShipWaveDataLoPtr),Y
        BEQ ReturnFromLoadingWaveData

        ; Set Y Pos to $90 if we have wave data in Bytes 8-9.
        LDA #$90
        LDY previousAttackShipIndexTmp
        STA upperPlanetAttackShipsYPosArray + $01,Y

ReturnFromLoadingWaveData
        RTS
```

Listing 1.15: "The sub-routine `SetInitialRandomPositionUpperPlanet` in `GetWaveDateForNewShip`. "

The first order of business is to call `PutRandomByteInAccumulatorRegister` which gets a random value and stores it in the accumulator.

```
SetInitialRandomPositionUpperPlanet
```

Since `A` can now contain anything from 0 to 255 ($00 to $FF) this needs to be adjusted to a meaningful Y-position value for the upper planet. So if we imagine `PutRandomByteInAccumulatorRegister` returned $85, we now do the following operations on it:

```
        JSR PutRandomByteInAccumulatorRegister
        AND #$3F
        CLC
```

First we do an `AND #$3F` with the value of $85 in `A`:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$85	1	0	0	0	0	1	0	1
$3F	0	0	1	1	1	1	1	1
Result	0	0	0	0	0	1	0	1

AND'ing $3F and $85 gives $05.

Our result is $05. The effect of the AND'ing here is to ensure that the random number we get back is between 0 and 63 rather than 0 and 255. Next we add $40 (decimal 64) to this result:

```
        AND  #$3F
        CLC
        ADC  #$40
```

This gives $45 and this is what we store as the initial Y position for the enemy.

You'll notice that the steps for `SetInitialRandomPositionLowerPlanet` are identical but with only the constant of the add value of $98 instead of $40. This is simply an additional offset to ensure that the Y position is lower on the screen for the initial position of the enemy on the lower planet.

We still haven't got into what Byte 7 is doing though. With an initial Y position determined, it looks like the intention was for Byte 7 to specify some adjustment to this value. But this looks like another bit of non-functioning game logic. If Byte 7 contains a value, the function will return early without any further adjustments. If it's zero it will then try Byte 9. If that's zero, it will return early. So the logic needs Byte 7 to be zero and Byte 9 to contain something for anything to happen. That's never the case, so the the adjustment never happens:

```
        STY  previousAttackShipIndexTmp
        ; Byte 7 ($06): Usually an update rate for the attack
    ships.
        LDY  #$06
        LDA  (currentShipWaveDataLoPtr),Y
        BNE  ReturnFromLoadingWaveDataEarly

        ; Byte 9 ($08): Default initiation Y position for the
    enemy.
        LDY  #$08
        LDA  (currentShipWaveDataLoPtr),Y
        BEQ ReturnFromLoadingWaveDataEarly

        LDA #$6C
        LDY previousAttackShipIndexTmp
```

Listing 1.16: ”An adjustment that never happens. Byte 7 and Byte 9 are never set in this way”

This is definitely some forgotten code. Byte 7 is elsewhere used in combination with Byte 8 and Byte 9 to define an alternate enemy mode for some levels where the ship will supplement any dead ships with alternate enemy types and attack patterns periodically.

1.0.6 Pointer Data

This happens in `MaybeSwitchToAlternateEnemyPattern` in `UpdateAttackShipDataForNewShip`.

```
MaybeSwitchToAlternateEnemyPattern
        ; Byte 7 ($06): Usually an update rate for the attack
    ships.
        LDY #$06
        LDA (currentShipWaveDataLoPtr),Y
        BEQ EarlyReturnFromAttackShipBehaviour

        DEC rateForSwitchingToAlternateEnemy,X
        BNE EarlyReturnFromAttackShipBehaviour

        LDA (currentShipWaveDataLoPtr),Y
        STA rateForSwitchingToAlternateEnemy,X

    ; Push the current ship's position data onto the stack.
        TXA
        PHA
        LDY indexIntoUpperPlanetAttackShipXPosAndYPosArray,X
        LDA upperPlanetAttackShipsXPosArray + $01,Y
        PHA
        LDA upperPlanetAttackShipsMSBXPosArray + $01,Y
        PHA
        LDA upperPlanetAttackShipsYPosArray + $01,Y
        PHA

        ; Are we on the top or bottom planet?
        TXA
        AND #$08
        BNE LowerPlanetAttackShipBehaviour
        ; We're on the upper planet.
        LDX #$02
ProcessAttackShipBehaviour
        JSR SetXToIndexOfShipThatNeedsReplacing
        BEQ ResetAndReturnFromAttackShipBehaviour

    ; Pop the current ship's position data from the stack and
    ; populate it into the new ship's position.
        LDY indexIntoUpperPlanetAttackShipXPosAndYPosArray,X
        PLA
        STA upperPlanetAttackShipsYPosArray + $01,Y
        PLA
        BEQ MSBXPosOffsetIzZero

        LDA attackShipsMSBXPosOffsetArray + $01,X
MSBXPosOffsetIzZero
    STA upperPlanetAttackShipsMSBXPosArray + $01,Y
        PLA
        STA upperPlanetAttackShipsXPosArray + $01,Y
        PLA
```

Listing 1.17: ”Byte 7 is used to periodically switch to an enemy mode defined by Bytes 8-9”

Byte 7 is used to drive the rate at which this routine switches over to the enemy data/-mode defined by Byte 8 and Byte 9.

```
        DEC rateForSwitchingToAlternateEnemy,X
        BNE EarlyReturnFromAttackShipBehaviour

        LDA (currentShipWaveDataLoPtr),Y
        STA rateForSwitchingToAlternateEnemy,X
```

Listing 1.18: ”`rateForSwitchingToAlternateEnemy` (Byte 7) is decremented and reloaded each time it reaches zero.”

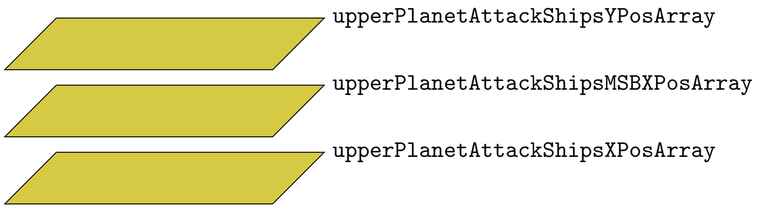
What this routine is going to do is replace the first dead ship it finds in the current wave with the wave data pointed to by Byte 8-9 and create a new enemy with the current ship's position with it.

First, we store the current ship's position. The way to do this is get the index (`Y`) for the current ship `X` and store each of the X and Y Position information into the accumulator first `A` and then push it onto the 'stack' (`PHA` which means 'push `A` onto the stack').

```
    ; Push the current ship's position data onto the stack.
```

```
        TXA
        PHA
        LDY indexIntoUpperPlanetAttackShipXPosAndYPosArray,X
        LDA upperPlanetAttackShipsXPosArray + $01,Y
        PHA
        LDA upperPlanetAttackShipsMSBXPosArray + $01,Y
        PHA
        LDA upperPlanetAttackShipsYPosArray + $01,Y
        PHA
```

When this has run the stack of accumulator values now looks like this:


upperPlanetAttackShipsYPosArray
upperPlanetAttackShipsMSBXPosArray
upperPlanetAttackShipsXPosArray

The stack after the code above has run with `upperPlanetAttackShipsXPosArray` at the top.

With our position data safely stashed away on the stack we now decide which planet we're on:

```
        ; Are we on the top or bottom planet?
        TXA
        AND #$08
        BNE LowerPlanetAttackShipBehaviour
```

If we're on the upper planet we use `SetXToIndexOfShipThatNeedsReplacing` look in the `activeShipsWaveDataHiPtrArray` for any ships that need replacing between positions `$02` and `$06`. If we don't find one, we return early:

```
        ; We're on the upper planet.
        LDX #$02
ProcessAttackShipBehaviour
        JSR SetXToIndexOfShipThatNeedsReplacing
        BEQ ResetAndReturnFromAttackShipBehaviour
```

If we do find one we can now pull (or 'pop') the positional data we stored away in the stack and assign that to the once-dead ship. First we use the index we retrieved to `X` to get the ship's index (`Y`) into the positional arrays:

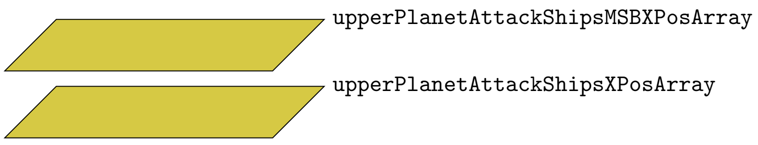
```
        ; Pop the current ship's position data from the stack and
        ; populate it into the new ship's position.
          LDY indexIntoUpperPlanetAttackShipXPosAndYPosArray,X
```

Then we pop the first positional item `upperPlanetAttackShipsYPosArray` from the top of the stack and store in the new ship's position in the array:

```
          PLA
          STA upperPlanetAttackShipsYPosArray + $01,Y
```

Listing 1.19: ”`PLA` remove the top item from the stack and stores it in `A`

The stack now looks like this, popping from the stack has the effect of removing the first item:


upperPlanetAttackShipsMSBXPosArray
upperPlanetAttackShipsXPosArray

Then we pop the rest of the items one by one and assign them to the new ship. We ignore the sprite's MXB offset if it is zero:

```
          PLA
          BEQ MSBXPosOffsetIzZero

          LDA attackShipsMSBXPosOffsetArray + $01,X
MSBXPosOffsetIzZero
        STA upperPlanetAttackShipsMSBXPosArray + $01,Y
          PLA
          STA upperPlanetAttackShipsXPosArray + $01,Y
          PLA
```

Listing 1.20: ”`PLA` remove the top item from the stack and stores it in `A`

Now that we have set up the positional data for the new enemy we load all its other features from the data pointed to by Byte 8-9:

```
          ; Byte 8 of Wave Data gets loaded now. Bytes 8 and 9
          ; contain the hi/lo ptrs to the alternate enemy data.
          LDY #$07
          JMP UpdateWaveDataPointersForCurrentEnemy
```

Let's take a closer look at this routine `UpdateWaveDataPointersForCurrentEnemy`.

What it does in this instance is take the address pointed to by Bytes 8 and 9 and load the data there using the routine `GetWaveDataForNewShip`. To be used in this way the values in Bytes 8 and 9 are combined and treated as an address in memory. For example if Byte 8 contains `$70` and Byte 9 contains `$13` they are treated as providing the address `$1370`. This is the location where the enemy data for `planet1Level8Data` is kept so that is what is loaded.

Planet	Level	Byte 7	Bytes 8-9
1	11	$03	smallDotWaveData
1	14	$03	planet1Level8Data
2	19	$0C	landGilbyAsEnemy
3	4	$04	gilbyLookingLeft
3	6	$04	planet3Level6Additional
4	19	$01	planet4Level19Additional
5	3	$01	planet5Level3Additional
5	5	$05	planet5Level5Additional
5	14	$06	llamaWaveData

Byte 7 : Whether a specific attack behaviour is used.
Bytes 8-9 : Lo and Hi Ptr for alternate enemy mode

Table 1.1: Actual use of Bytes 7, 8, and 9. Note that the value in Byte 7 doesn't matter, as long as it's non-zero.

1.0.7 Enemy Behaviour

1.0.8 Level Movement Data

Notes & References

[1] **What is a Medal game?** This is not a typo! A medal game is played with metal coins. The most famous ones are ”pusher games” where the player must drops coins in a platform system. Each platform moves back and forth as automated brooms. The goal is to push coins groups past the edge of the final platform where they are rewarded to the player.

[2] **What is a Planner?** They were the top decision maker in a Japanese game dev team. Responsible for giving directions and making game design decisions, all other members of the team reported to them. There was usually a single Planner in charge (like Poo on 1943: The Battle of Midway) but there could be two like in Street Fighter II where both Akira Nishitani (Nin) and Akira Yasuda (Akiman) were in charge.

[3] **”1942 Final Review Team Arcade”** (by Tyler Huberty, Greg Nazario, Isaac Simha, link, 2012-09-12.

[4] **”Computer Gamer Magazine #4”** (”Coin-Op Connection” article, link, 1985-07.

[5] **”Questionable figures”**, The figures of ”two years and five million dollars” should be taken carefully. These numbers were found on a Forgotten Worlds flyer (**??**) which also mentioned three Motorola 68000s whereas the final product only included one. 1989.

[6] **”The story of the 3dfx”** (by Fabien Sanglard), link, 2019-04-04.

[7] **”The Sound of Innovation: Stanford and the Computer Music Revolution”** (by Andrew J. Nelson), ISBN: 978-0262028769. 2015-03-06

[8] **”The birth of Chun-Li”** (Akiman for Archipel), link, 2018-02.

[9] **”Computer Speed Claims 1980 to 1996”** (Roy Longbottom), link.

[10] **”Les grands noms du jeu video, Yoshihisa Kishimoto - Enter the Double Dragon”** (Florent Gorges for Editions PixNlove), link, 2012-07-05.

[11] **"Akiman's Twitter"** (akiman), post 1, post 2, post 3.

[12] **"Top 10 Highest-Grossing Arcade Games of All Time"** (Jaz Rignall for usgamer.net) (200,00 units: SF2 WW sold 60,000 while SF2 CE sold 140,000), link, 2016-01-01.

[13] **"World of Warcraft Leads Industry With Nearly $10 Billion In Revenue"** (Jonathan Leack for gamerevolution.com), link, 2014-26-01.

[14] **"Interview with Noritaka Funamitsu"** (Retro magazine), part 1, part 2, part 3.

[15] **"Mame CPS-1 video driver"** (mame source code), link, 2008-04-11.

[16] **"Mame CPS-1 driver"** (mame source code), link, 2008-04-11.

[17] **"Kabuki z80 encryption"** (mame source code), link, 2008-04-11.

[18] **"Early CAPCOM Arcade Games FGPA"** (Jose Tejada), link, 2020-08-05.

[19] **"Genesis mode H40"** The vertical and horizontal rates in H40 are not the numbers we would get if we were to inject the dot-clock, number of dots, and number of lines in the formulas. This is because the Genesis designers wanted to have the same rate in H32 and H40 modes (59.92 Hz). The dot-clock slows down to 5.37MHz for 28 dots during HBLANK, resulting in 59.92 Hz VSYNC and 15,700 KHz HSYNC (Conversation with Upsilandre).

[20] **"Dot clock rates"** (pineight.com), link.

[21] **"Final Fight Developer's Interview"** (capcom.com), link, 2019-02-08.

[22] **"Street Fighter II Developer's Interview"** (capcom.com), link, 2018-11-21.

[23] **"Capcom Activity Report: Akira Yasuda part 1"** (capcom.com), link, 2016-03-31.

[24] **"Capcom Activity Report: Akira Yasuda part 2"** (capcom.com), link, 2016-04-04.

[25] **"Capcom, A captive audience"** (Robin Hogg & Dominic Handy for The Games Machine, Issue #19), link, 1989-06-01.

[26] **"Yoshiki Okamoto interview"** (Gamest Magazine #38), link, 1989-10-01.

[27] **"Final Fight arcade 2 players"** (arronmunroe), link (Use ',' and '.' to move frame by frame) 2013-10-12.

[28] **"DL-0921 (CPS-B-21) Video Signals Generation"** (Loïc Petit), link, 2020-11-29.

[29] **"DL-0921 (CPS-B-21) Security Scheme"** (Loïc Petit), link.

[30] **"Capcom CPS1"** (Eduardo Cruz), part 1, part 2, part 3, 2015-04-16.

[31] **”Capcom Kabuki CPU”** (Eduardo Cruz), intro, part 1, part 2, part 3, part 4, part 5, 2014-11-16.

[32] **”CAPCOM CPS1 Reverse Engineering”** (Eduardo Cruz), link, 2015-06-15.

[33] **”CPS1 Project Update”** (Eduardo Cruz), link, 2015-09-19.

[34] **”Chip Hall of Fame: Motorola MC68000 Microprocessor”** (spectrum.ieee.org), link, 2017-06-30.

[35] **”Instruction prefetch on the Motorola 68000 processor”** (Jorge Cwik), link, 2005.

[36] **”CPS-2 Rebirth !!!!”** (cps2shock.retrogames.com), link, 2003-04-23.

[37] **”We now have a non encrypted version of the encrypted SFZ program ROM”** (cps2shock.retrogames.com), link, 2000-12-32.

[38] **”Mame CPS-2 Driver, keys (cps2crpt.c)”**, link.

[39] **”Street fighter 2 WW glitch invisible dhalsim”** (youtube.com Error1), link, 2010-09-22.

[40] **”Blending Worlds With Music: Interview With Composer Yoko Shimomura”** (otaquest.com), link, 2019-12-26.

[41] **”Programmer’s Guide to Yamaha YMF 262/OPL3 FM Music Synthesizer”** (Vladimir Arnost), link, 2019-12-26.

[42] **”CPS-B Number”** (tim for arcadecollection.com), link.

[43] **”The Untold History of Japanese Game Developers Volume 1 (Interview: Koichi Yotsui)”** (John Szczepaniak), 2015-11-04.

[44] **”Game Maestro #4”**, link.

[45] **”Street Fighter 2: Oral History”** (Matt Leone), link. 2014-02-03.

[46] **”Blending Worlds With Music: Interview With Composer Yoko Shimomura”** (OTAQUEST Editor), link. 2019-12-26.

[47] **”BEEP! Megadrive magazine: The Women of Game Making”** (translated shmuplations.com), link. 1990-10.

[48] **”Unfinished Strider Conversion”** (Shoestring), link. 2016-02-17.

[49] **”How to Phoenix a CPS 2 PCB”** (Joe Bagadonuts), link. 2015-05-18.

[50] **”Dialogic ADPCM Algorithm”** (Dialogic Corporation), link. 1988.

[51] **"Street Fighter 2 Manual"** (Capcom Corporation), link. 1992.

[52] **"Street Fighter 2: The AI engine"** (Ben Torkington), link. 2017-1-20

[53] **"X68000 Sprite management"** (Koichi Yoshida), link. 2021-02-25

[54] **"How To Make Capcom Fighting Characters"** (Akiman, Kiki, Bengus), ISBN: 978-1772941364. 2020-010-20

[55] **"Akiman, 2003 Interview from Capcom Design Works"** (Akiman, translated shmuplations), link. 2003

[56] **"A Talk Between the Creators of Street Fighter and Fatal Fury: KOF"** (Yoshiki Okamoto and Takashi Nishiyama), link. 2021-08-09

[57] **"Street Fighter II Complete File"** (Capcom edition), ISBN: 978-4257090014. 1992-11-15

[58] **"Shoryuken..! The music of Street Fighter II"** (909originals), link 2021-21-02

[59] **"The CPS-1 SDK, a.k.a CAT-A"** (Akiman), link 2018-07-01

[60] **"The CPS-1 SDK, a.k.a CAT-A: Additional details"** (Takenori Kimoto (a.k.a KimoKimo)), link 2018-07-02

[61] **"Private View:** 月刊電脳倶楽部 **(GEKKAN DENNŌ CLUB)"** (Ted Danson), link 2015-06-25

[62] **"MSM6295 datasheet"** (by OKI), link

[63] **"Sony SMC-70 Microcomputer"** (by Ahm), link 2011-05-19

[64] **"Capcom – Retrospective Interview"** (by https://shmuplations.com/), link 1991

[65] **"Street Fighter II Interview Soundtrack OST"** (Yoko Shimomura), link 2017-10-28

[66] **"Diggin' In The Carts, Hidden Levels"** (conversation Yoko Shimomura with Manami Matsumae), link 2014-09-23

[67] **"Street Fighter II Platinum Source Code"** (Ben Torkington), link 2021-10-10

[68] **"Japan's Technical Standards: Implications for Global Trade and Competitiveness"** (John Mcintyre), ISBN: 978-1567200539. 1997-02-28

[69] **"Human68k Manual"** (gamesx.com), link 2019-08-27

[70] **"Le x68000 et la supériorité japonaise"** (upsilandre), link 2020-12-04

[71] **"X68000 Perfect Catalogue"** (by G-Walk), ISBN: ISBN4867171018. 2020-10-27

From the preface:

Before the era of overpowered PCs and home consoles, there was a time when video-game enthusiasts could only experience the very best and the most challenging in places called "arcades".

In these locations, players of all ages and origins gathered to take their passion to a level no consumer grade hardware could.

The arcades of the early '90s were a highly competitive environment where publishers only had a few seconds to catch a player's attention, and more importantly their quarters. It was during that time that a young company named Capcom managed to elevate itself above the competition and turn itself into an icon.

This book is an engineering love letter to the platform that allowed this metamorphosis. If you have always wanted to learn about the machine behind the legendary CPS-1 titles Street Fighter II, Ghouls 'n Ghosts, and Final Fight, the "Book of CP-System" is for you.

Inside, you will find the hardware of the CPS-1 described and explained in excruciating detail. The software is also covered with a fully detailed modern pipeline, turning code and assets into ROMs.

Jump in and discover a world of one hundred explanatory illustrations, sprinkled with typos and broken English to remind you this isn't just a dream!